INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

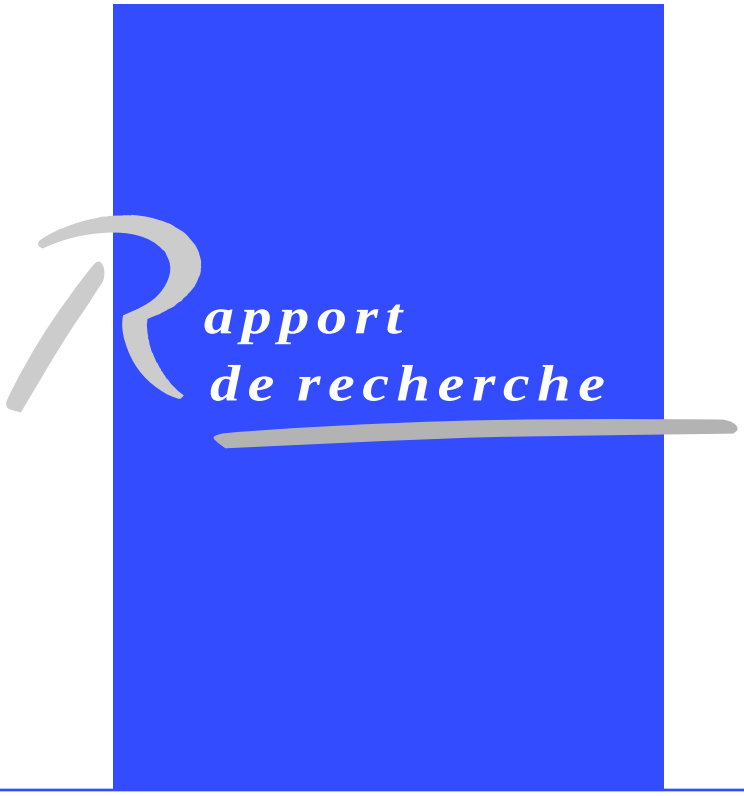
A Tracer Driver to Enable Concurrent Dynamic Analyses

Ludovic Langevine and Mireille Ducassé

N°5611

Juin 2005

Systèmes symboliques



ISSN 0249-6399 ISRN INRIA/RR--5611--FR+ENG

A Tracer Driver to Enable Concurrent Dynamic Analyses

Ludovic Langevine * and Mireille Ducassé †

Systèmes symboliques
Projets Lande et Contraintes

Rapport de recherche n ° 5611 — Juin 2005 — 28 pages

Abstract: Tracers provide users with useful information about program executions. In this report, we propose a "tracer driver", from a single tracer, it provides a powerful front-end for multiple dynamic analysis tools, while limiting the overhead of the trace generation. The tracer driver can be used both synchronously and asynchronously. The relevant execution events are specified by flexible event patterns and a large variety of trace data can be given either systematically or "on demand". The proposed tracer driver has been designed and experimented in the context of constraint logic programming, within GNU-Prolog. Its principles are, however, independent of the traced programming language. Experimental measures show that the flexibility and power of the described architecture are also the basis of reasonable performances.

Key-words: Software Engineering, Debugging , Execution Monitoring, Execution Tracing, Execution Visualization Programming Environment, Constraint Logic Programming

(Résumé : tsvp)

This work has been partially supported by the French RNTL project OADymPPaC, http://contraintes.inria.fr/OADymPPaC/, by the ERCIM program and by SICS in Sweden.

* Ludovic.Langevine@inria.fr
† Mireille.Ducasse@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un pilote de traceur pour permettre des analyses dynamiques concurrentes

Résumé : Les traceurs fournissent aux utilisateurs des informations utiles sur les exécutions de programmes. Dans ce rapport, nous proposons un « pilote de traceur », à partir d'un même traceur, il fournit un puissant front-end à de multiples analyses dynamiques, tout en limitant le sur-coût de la génération de la trace. Le pilote de traceur peut être utilisé à la fois de manière synchrone et asynchrone. Les événements d'exécution pertinents sont spécifiés par des motifs flexibles d'événements et une grande variété de données de trace peut être fournie soit systématiquement doit à la demande. Le pilote de traceur proposé a été conçu et expérimenté dans le contexte de la programmation logique par contraintes, dans GNU-Prolog. Ses principes sont cependant indépendants du langage tracé. des mesures expérimentales montrent que la flexibilité et la puissance de l'architecture décrite sont aussi la raison de performances raisonnables.

Mots-clé : Génie logiciel, débogage, supervision d'exécutions, trace d'exécutions, visualisation d'exécutions, environnement de programmation, programmation logique par contraintes

1 Introduction

Dynamic program analysis consists in analyzing program executions. It is generally acknowledged that dynamic analysis is complementary to static analysis, see for example the discussion of Ball [2]. Dynamic analysis tools include, in particular, tracers, debuggers, monitors and visualizers.

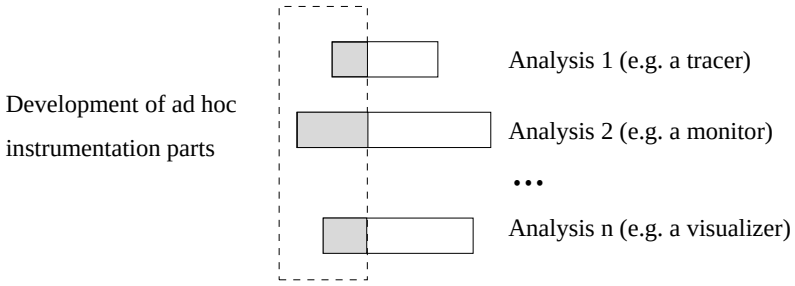


Figure 1: The usual case: all dynamic analysis tools implement a dedicated instrumentation part

In order to be able to analyze executions, some data must be gathered and some sort of instrumentation mechanisms must be implemented. The state-of-the-practice, illustrated by Fig. 1, is to re-implement the instrumentation for each new dynamic analysis tool. The advantages are, firstly, that the instrumentation is naturally and tighly connected to the analysis, and secondly, that it is specialized for the targeted analysis and produces relevant information. The drawback, however, is that this implementation requires, in general, much tedious work and discourages people to develop dynamic analysis tools.

1.1 A Tracer Driver to Efficicently Share Instrumentations

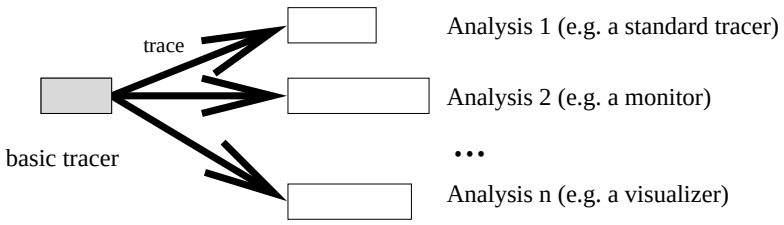


Figure 2: The "generate-and-dump" approach: the instrumentation part is shared but the amount of data is huge

In this article we suggest that standard tracers can be used to give information about executions to several dynamic analysis tools. Indeed, Harrold et al. have shown that a

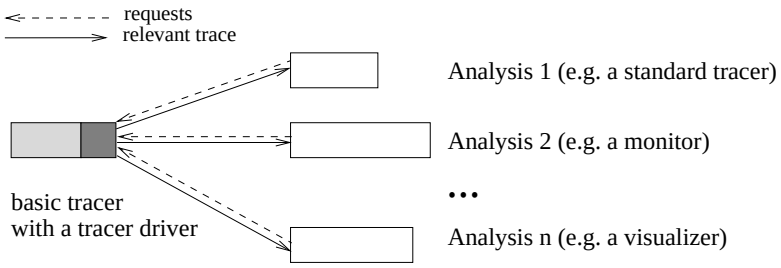


Figure 3: Our supervised generation approach: with a tracer driver only the relevant part of the execution information is generated

trace spectrum, consisting of the sequence of program statements traversed as the program executes, subsumes a number of interesting other spectra such as the set of conditional branches or the set of paths [11]. However, the separation of the extraction part from the analysis part cannot be done without care. Indeed, as illustrated by Fig. 2, if execution information is systematically generated and dumped to the analysis tools, the amount of information that flows from the tracer to the analysis modules can be huge, namely several gigabytes for a few seconds of execution. Whether the information transit through a file, a pipe, or even main memory, writing such an amount of information takes so much time that the tools are not usable interactively. It is especially critical for debugging, even when it is automated, because users need to interact in real-time with the tools.

Reiss and Renieris propose to encode and compact the trace information [17]. Their approach is used in a context where multiple tracing sources send information to the same analysis module. In this article, we propose another approach, more accurate when a single source sends information to (possibly) several analysis modules. As illustrated by Fig. 3, we have designed what we call a "tracer driver", whose primary function is to filter the data on the fly according to requests sent by the analysis modules. Only the necessary trace information is actually generated. This often drastically reduces the amount of trace data, and significantly improves the performance.

Therefore, the instrumentation module is shared among several analysis tools and there is very little slowdown compared to the solution where each analysis has its dedicated instrumentation. From a single tracer, the tracer driver provides a powerful front-end for multiple dynamic analysis tools while limiting the overhead of the trace generation. The consequence is that specifying and implementing dynamic analysis tools is much easier, without negative impact on the end-user.

1.2 Tracer and Analyzers Interactions

In the following, we call "analyzer" a module which is connected to a tracer. In its simplistic form the analyzer is only the standard output or a file in which traces are written by a primitive tracer. For standard tracing tools, the analyzer handles the interaction once the

execution is stopped at interesting points (called "breakpoints"). It shows some trace information and reacts on users' commands. More sophisticated debugging tools exhibit more sophisticated analyzers. A trace querying mechanism with a database flavor can be plugged to a tracer and let users investigate executions in a more thorough way. This has been done for example for C in the Coca tool [7]. A real database can even be used if on the fly performance is not a big issue. This has been done for a distributed system in Hy^+ [5]. Monitoring tools can be connected to tracers in order to supervise executions and collect data. For example, the EMMI tool, for Icon, is able to detect some programming mistakes [13]. A number of visualization tools use traces to generate graphical views. An example is DAQV which offers graphical clients implementing different views of high performance Fortran [10].

The interaction modes between the tracer and the analyzers exhibited by the previous examples are all different and specific. For primitive tracers and simple visualization, the tracer simply outputs trace information into a given channel. There is no synchronization between the tracer and the analyzer. Standard tracers and trace query systems output trace information and get user requests in a totally synchronized way. Monitors process the trace information on the fly in a fully synchronized way. At present, all these tools are disjoint and difficult to merge. Therefore, further mechanisms are required in order to share a tracer among analyzers of different types.

Our tracer-driver includes such mechanisms. It enables different interaction modes between a tracer and analyzers to be integrated in one single tool. This has several advantages. Firstly, users do not need to switch tool to achieve different aims. They use a unique tool to trace, debug, monitor and visualize executions. Secondly, integrating all the possible usages results in a more powerful tool than the mere juxtaposition of different tools. For example, one can, in parallel, check for known bug patterns, and collect data for visualization. Whenever a bug is encountered the tool can switch to a synchronized debugging session, using the already collected visualization data. The visualization tool can also change the granularity of the collected data depending on the current context.

1.3 Contributions

The contributions of this article are threefold. Firstly, it justifies the need for a tracer driver in order to be able to efficiently integrate several dynamic analyses within a single tool. In particular, it emphasizes that both synchronous and asynchronous communications are required between the tracer and the analyzer. Secondly, it describes in breadth and in some depth the mechanisms needed to implement such a tracer driver: 1) the patterns to specify what trace information is needed, 2) the language of interaction between the tracer driver and the analyzers and 3) the mechanisms to efficiently filter trace information on the fly. Lastly, experimental measurements show that this architecture increases the trace accuracy and speeds up trace generation and communication.

In the following, Section 2 gives an overview of the tracer driver and in particular the interactions it enables between a tracer and an analyzer. Section 3 specifies the nature of patterns. Section 4 lists the requests that an analyzer can send to our tracer and how they

are taken into account. Section 5 describes in detail our filtering mechanism and its implementation. Section 6 discusses the requirements on the tracer for the overall architecture to be efficient. Section 7 gives experimental results and shows the efficiency of the tracer driver mechanism. Section 8 discusses related work.

2 Overview of the tracer driver

This section presents an overview of the tracer driver architecture and, in particular, the interactions it enables between a tracer and analyzers. The tracer and the analyzers are run at the same time.

As already mentioned in the introduction, both synchronous and asynchronous interactions are necessary between the tracer and the analyzers. On the one hand, if analyzers need to get complements of information at some events, it is important that the execution does not proceed until the analyzers have decided so. On the other hand, if the analyzers only want to collect information there is no need to block the execution.

An execution trace is a sequence of observed execution events that have attributes. The analyzers specify the events to be observed by the means of *event patterns*. An event pattern is a condition on the attributes of an event (see details in Section 3). The tracer driver manages a base of active event patterns. Each execution event is checked against the set of active patterns. An event matches an event pattern if and only if the pattern condition is satisfied by the attributes of this event.

An *asynchronous pattern* specifies that, at matching trace events, some trace data are to be sent to analyzers without freezing the execution. A *synchronous pattern* specifies that, at matching trace events, some trace data are to be sent to analyzers. The execution is frozen until the analyzers order the execution to resume. An *event handler* is a procedure defined in an analyzer, that is called when a matching event is encountered.

Fig. 4 illustrates the treatment of the two types of patterns. The execution is presented as a sequence of elementary blocks (the execution events). An analyzer mediator gathers the patterns requested by the analyzers and dispatches the result sent by the tracer driver (see detailed description Section 4). At each trace event, the tracer driver is called to filter the event. If the current event does not match any of the active specified patterns, the execution goes on (events i, $i+1$, $i+2$, $i+4$, $i+6$). If the current event matches an active pattern, some trace data are sent to the analyzer mediator (events $i+3$, $i+5$). If the matched pattern is asynchronous the data is processed by the relevant analyzer in an asynchronous way (event $i+3$). If the pattern is synchronous the execution is frozen, waiting for a query of the analyzer (event $i+5$). The analyzer processes the sent data and can ask for more data about the state of the execution. The tracer driver can retrieve useful data about the execution state and send them to the analyzer "on demand". The analyzer can also request some modifications of the active patterns: add new patterns or remove existing ones. When no analyzer has any further request to make about the current event, the analyzer mediator sends the resuming command to the tracer driver (*go* command). The tracer then resumes the execution until the next matching event.

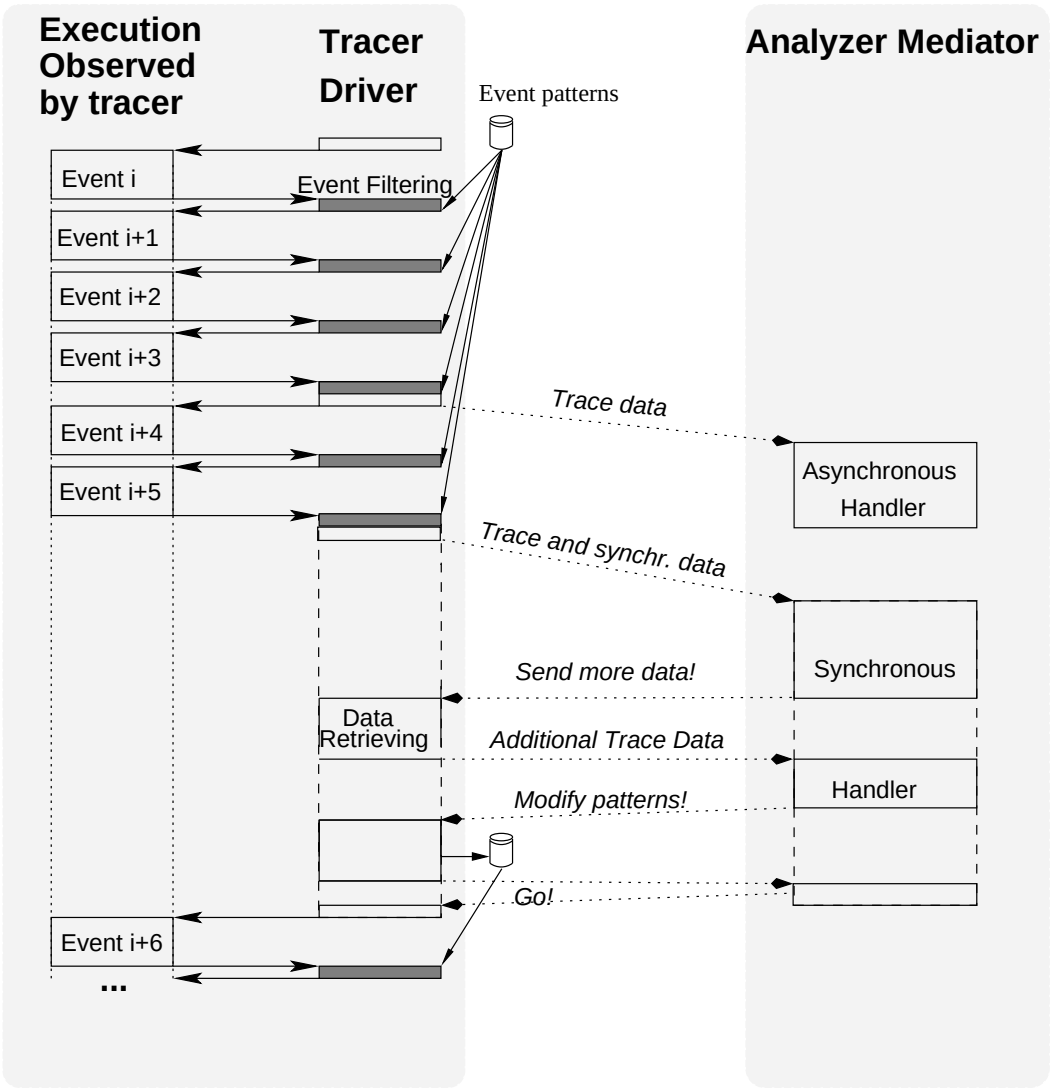


Figure 4: Asynchronous and synchronous interactions between the tracer and the analyzer mediator

The architecture enables the management of several active patterns. Each pattern is identified by a label. A given execution event may match several patterns. When sending the trace data the list of (labels of) matched patterns is added to the trace. Then, the analyzer mediator calls a specific handler for each matched pattern and dispatches relevant

trace data to it. If at least one matched pattern is synchronous, the analyzer mediator waits for every synchronous handler to finish before sending the resuming command to the tracer driver. From the point of view of a given event handler, the activation of other handlers on the same execution event is transparent.

This article emphasizes more the tracer driver than the analyzer mediator. On the one hand, the design and implementation of the tracer driver is critical with respect to response time. Indeed it is called at each event and executions of several millions of events (see section 7) are very common. Every overhead, even the tiniest, is therefore critical. On the other hand, the implementation of the analyzer mediator is much less critical because it is called only on matching events. Furthermore, its implementation is much easier.

3 Event patterns

As already mentioned, an event pattern is a condition on the attributes of events. It consists of a first order formula combining elementary conditions on the attributes. This section summarizes the format of the trace events, specifies the format of the event patterns and gives examples of patterns.

3.1 Trace events

The actual format of the trace events has *no influence* on the tracer driver mechanisms. The important issue is that events have attributes and that some attributes are specific to the type of events. The trace format that we use is dedicated to constraint programming over finite domains, but the pattern language is *independent of the traced language.*

This section summarizes the format. Its aim is to help the readers who would want to understand the details of the examples in the remaining of the article.

A constraint program manipulates variables and constraints on these variables. Each variable has a *domain*, a finite set of possible values. The aim of a constraint program is to find a valuation (or the best valuation, given an objective function) of the variables such that every constraint is satisfied. To do so, constraint solvers implement numerous algorithms coming from various research areas, such as operation research. There are 14 possible event types in the tracer we use. Each event has common and specific attributes. The common attributes are: the event type, a chronological event number, the depth of the current node in the search-tree, the solver state (domains, constraint store and propagation queue), and the *user time* spent since the beginning of the execution. The specific attributes depend on the port. They are not detailed here. Whenever they will be used in the following they will be paraphrased.

Fig. 5 presents the beginning of a trace of a toy program in order to illustrate the events described above. This program,
`fd_element(I, [2,5,7],A), (A#=I ; A#=2)`, specifies that `A` is a finite domain variable which is in $\{2, 5, 7\}$ and `I` is the index of the value of `A` in this list; moreover `A` is either

```
1 newVariable   v1=[0-268435455]
2 newVariable   v2=[0-268435455]
3 newConstraint c1 fd_element([v1,[2,5,7],v2])
4 reduce c1 v1=[1,2,3] W=[0,4-268435455]
5 reduce c1 v2=[2,5,7] W=[0-1,3-4,6,8-268435455]
6 suspend c1
```

Figure 5: A portion of trace

pattern ::= ***label*: when** *evt_pattern* *op_synchro* *action_list*
op_synchro ::= **do** | **do_synchro**
action_list ::= *action* , *action_list* | *action*
action ::= **current**(*list_of_attributes*) | **call**(*procedure*)

evt_pattern ::= *evt_pattern* **or** *evt_pattern* **(1)**
| *evt_pattern* **and** *evt_pattern* **(2)**
| **not** *evt_pattern* **(3)**
| (*evt_pattern*) **(4)**
| *condition* **(5)**
condition ::= ***attribute*** *op2* ***value*** | *op1*(***attribute***) | **true**
op2 ::= < | > | = | \= | >= | =< | **in** | **notin**
| **contains** | **notcontains**
op1 ::= **isNamed**

Figure 6: Grammar of event patterns

equal to `I` or equal to 2. The second alternative is the only feasible one. The trace can be read as follows. The first two events are related to the introduction of two variables `v1` and `v2`, corresponding respectively to `I` and `A`. In Gnu-Prolog, variables are always created with the maximum domain (from 0 to $2^{28}-1$). Then the first constraint is created: `fd_element` (event #3). This constraint makes two domain reductions (events #4 and #5): the domain of the first variable (`I`) becomes $\{1,2,3\}$ and the domain of `A` becomes $\{2,5,7\}$, the only consistent values so far. After these reductions, the constraint is suspended (event #6). The execution continues and finds the solution (`A=2`, `I=1`) through 32 other events not shown here.

3.2 Patterns

We use patterns similar to the path rules of Bruegge and Hibbard [4]. Fig. 6 presents the grammar of patterns. A pattern contains four parts: a label, an event pattern, a synchronization operator and a list of actions. An event pattern is a composition of elementary conditions using logical conjunction, disjunction and negation. It specifies a class of execution event. A synchronization operator tells whether the pattern is asynchronous (**do**) or synchronous (**do_synchro**). An action specifies either to ask the tracer driver to collect

```
visu_cstr:
  when port = post
  do current(cstr=C and cstrRep=Rep
              and varC(cstr)=VarC),
     call new_cstr(C, Rep, VarC)
visu_prop:
  when port = reduce and isNamed(var)
         and (not cstrType='assign')
         and delta notcontains [maxInt]
  do current(cstr=C and var=V),
  call spy_propag(C,V)
symbolic:
  when port in [reduce,suspend]
       and (cstrType = 'fd_element_var'
           or cstrType = 'fd_exactly')
  do_synchro call symbolic_monitor
```

Figure 7: Examples of patterns for visualization and monitoring

attribute values (**current**(*list_ of_ attributes*)), or to ask the analyzer to call a procedure **call**(*procedure*). Note that the procedure is written in a language that the analyzer is able to execute. This language is independent of the tracer driver. An elementary condition concerns an attribute of the current event.

There are several kinds of attributes. Each kind has a specific set of operators to build elementary conditions. For example, most of the common attributes are integer (chrono, depth, node label). Classical operators can be used with those attributes: equality, disequality ($\neq$), inequalities ($<$, $\leq$, $>$ and $\geq$). The *port* attribute is the type of the current event. It has a small set of possible values. The following operators can be used with the port attribute: equality and disequality ($=$ and $\neq$) and two set operators, **in** and **notin**. Constraint solvers manipulate a lot of constraints and variables. Often, a trace analysis is only interested in a small subset of them. Operators **in** and **notin**, applied to identifiers of entities or name of the variables, can specify such subsets. Operators **contains** and **notcontains** are used to express conditions on domains. This set of operators is dedicated to the type of execution we trace. It could be extended to cope with other types of attributes.

3.3 Examples of patterns

Fig. 7 presents three patterns that can be activated in parallel. The first two patterns are visualization oriented: the first one requests the trace of each constraint-posting with the identifier of the constraint (a unique integer), its representation (the name of the constraint with its parameters) and the list of the involved variables (*varC*). Fig. 8 gives two screenshots of visualizations which are built using such patterns[1].

The second pattern requests the trace of all the domain reductions made by constraints that do not come from the assignment procedure and that do not remove the maximal integer

[1]The pictures are generated by Pavot a tool developed at INRIA Rocquencourt.

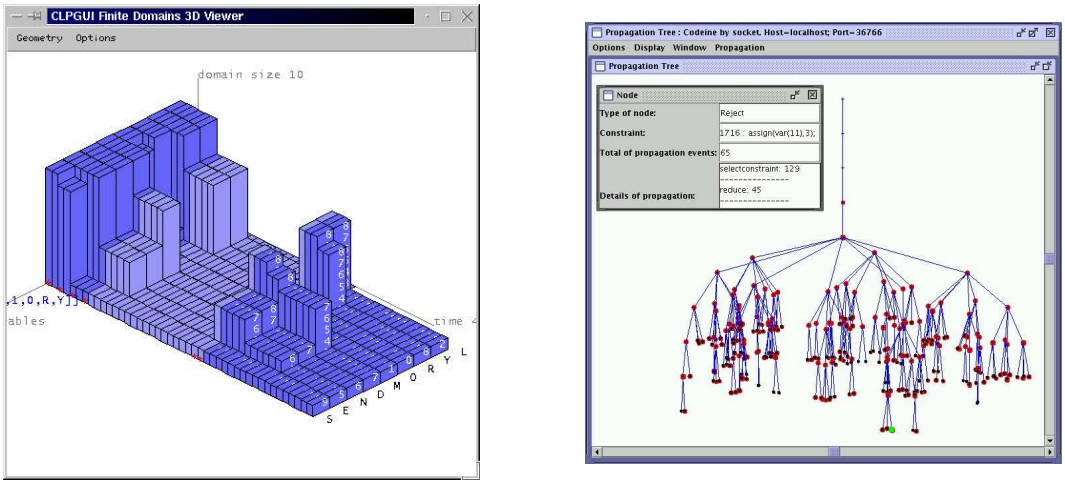


Figure 8: Two trace-based views based on the first pattern of Figure 7.

value. It stores the reducing constraint and the reduced variables. Those data can be used to compute some statistics and to visualize the impact of each constraint on its variables. Those two patterns are asynchronous: the requested data are sufficient for the visualization and the patterns do not have to be modified.

The last pattern is more monitoring-oriented: it freezes the execution at each domain reduction made by a symbolic constraint such as *element* (on variables) or *exactly*. This pattern allows the monitoring of the filtering algorithms used for these two constraints.

4 Analyzer mediator

The analyzer mediator processes the trace: it specifies to the tracer driver what events are needed and may execute specific actions for each class of relevant events. The mediator can supervise several analyses at a time. Each analysis has its own purpose and uses specific pieces of trace data. The independence of the concurrent analyses is ensured by the mediator that centralizes the communication with the tracer driver and dispatches the trace data to the ongoing analyses.

When tracer driver and mediator are synchronized, the requests that an analyzer can send to the driver are of three kinds. Firstly, the analyzer can ask for additional data about the current event. Secondly, the analyzer can modify the event patterns to be checked by the tracer driver. These patterns are called *active* in the following. Thirdly, the analyzer can notify the end of a synchronous session.

Primitive `current` specifies a list of event attributes to retrieve in the current execution event. The tracer retrieves the requested pieces of data. It sends the data as a list of pairs

```
step :-
  reset,
  add([step:when true
     dosynchro call(tracer_toplevel)]),
  go.

skip_reductions :-
  current(cstr = CId and port = P),
  reset,
  (  P == awake
  -> add([sr:when cstr = CId and port
    in [suspend,reject,entail]
    dosynchro call(tracer_toplevel)]),
  ;  add([step:when true
    dosynchro call(tracer_toplevel)])),
  go.
```

Figure 9: Implementation of two tracing commands

(*attribute*, *value*). `reset` deletes all the active event patterns and their labels. Primitive `remove` deletes the active patterns whose labels are specified in the parameter. Primitive `add` inserts in the active patterns, the event patterns specified in the parameter, following the grammar described in Figure 6. Primitive `go` notifies the tracer driver that a synchronous session is finished. The traced execution will be resumed.

Fig. 9 illustrates the use of the primitives to implement two tracing commands. Command `step` enables to go to the very next event. It simply resets all patterns and adds one which will match any event and call, in a synchronous way, the tracer toplevel. Command `skip_reductions` enables to skip the details of variable domain reductions when encountering the awakening of a constraint. It first checks the current port, if it is `awake` it asks to go to the suspension of this constraint. There, the user will, for example, be able to check the value of the domains after all the reductions. If the command is called on an event of other type it simply acts as `step`.

5 Filtering mechanism

This section describes in detail the critical issue of the filtering mechanism. At each execution event, it is called to test the relevance of the event with respect to the active patterns. Notice that the execution of a program with constraints can lead to several millions of execution events per second. Therefore, the efficiency of the event filtering is a key issue.

In the following, we first describe the algorithm of the tracer driver. Then we specify the automata which drive the matching of events against active patterns. We discuss some specialisation issues. We give some details about the incremental handling of patterns. Lastly, we explain that event attributes are computed only upon demand.

```
proc tracerDriver(P : set of active patterns)
  tagged ← ∅
  for each p ∈ P do
   if match(p) then tagged ← tagged ∪ {p}
  end for
  T ← {requested_data(p) | p ∈ tagged}
  send_trace_data(T, label(tagged))
  if synchronous(tagged) ≠ ∅ then
    notify(synchronous(tagged))
    repeat
      request ← receive_from analyzer()
      execute(request)
    until request = go
  end if
end proc
```

Figure 10: Algorithm of the Tracer Driver

5.1 Tracer driver algorithm

When an execution event occurs, the tracer is called. The tracer collects some data to maintain its own data structures and then calls the tracer driver. The algorithm of the tracer driver is given in Fig. 10. The filtering mechanism can handle several active event patterns. For each pattern, if the current event matches the pattern the latter is tagged as activated. Whatever the (matching) result is, the next pattern is checked. When no more patterns have to be checked, the tagged patterns are processed. The union of requested pieces of data is sent as trace data with the labels of the tagged patterns. If at least one synchronous pattern is tagged, a signal is sent to the analyzer and the tracer driver waits for requests coming from the analyzer and processes them until the **go** primitive is sent by the analyzer.

5.2 Pattern automata

The matching of an event against a pattern is driven by an automaton where each state is labeled by an elementary condition with two possible transitions: true or false. The automaton has two final states, **true** and **false**. If the **true** state is reached, the event is said to match the pattern. Each automaton results of the compilation of an event pattern. This compilation is specified by the attributed grammar of Fig. 11. The final automaton of the expression is a composition of sub-automata that express sub-expressions. The attributes of the grammar are *entry* (the entry point of the automaton, or of a sub-automaton), $ifTrue$ (transition if true) and $ifFalse$ (transition if false). *entry* is synthesized, $ifTrue$ and $ifFalse$ are inherited. **true** and **false** are the two final states. This grammar is inspired by the grammar to evaluate boolean expressions in imperative languages. It minimizes the number of conditions to check [20]. At elementary conditions, new_node$(C, E.IfTrue, E.IfFalse)$ creates

(0) $E \rightarrow E'$
$E.entry \leftarrow E'.entry$
$E'.ifTrue \leftarrow$ **true**
$E'.ifFalse \leftarrow$ **false**

(1) $E \rightarrow E_1$ **or** E_2
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifTrue$
$E_1.ifFalse \leftarrow E_2.entry$
$E_2.ifTrue \leftarrow E.ifTrue$
$E_2.ifFalse \leftarrow E.ifFalse$

(2) $E \rightarrow E_1$ **and** E_2
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E_2.entry$
$E_1.ifFalse \leftarrow E.ifFalse$
$E_2.ifTrue \leftarrow E.ifTrue$
$E_2.ifFalse \leftarrow E.ifFalse$

(3) $E \rightarrow$ **not** E_1
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifFalse$
$E_1.ifFalse \leftarrow E.ifTrue$

(4) $E \rightarrow$ (E_1)
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifTrue$
$E_1.ifFalse \leftarrow E.ifFalse$

(5) $E \rightarrow C$
$E.entry \leftarrow$
new_node($C, E.IfTrue, E.IfFalse$)

Figure 11: Attributed grammar to generate pattern automata

a node where the condition will be checked. If it is true its $ifTrue$ continuation will be connected to the $ifTrue$ continuation of the previous node, otherwise its $ifFalse$ continuation will be connected to the $ifFalse$ continuation of the previous node. Examples of automata are given in section 5.4.

5.3 Specialization thanks to the port

As seen in Section 3, the port is a special attribute since it denotes the *type* of the execution event being traced. A port corresponds to specific parts of the solver code where a call to a function of the tracer has been hooked. For example, in Codeine, there are 4 hooks for the reduce port, embedded into 4 specific functions that make domain reduction in 4 different ways. Furthermore, specific attributes depend on the port. As a consequence, the port is central in the pattern specification. For most of them, a condition on the port will be explicit. When an event occurs, it is useless to call the tracer driver if no pattern is relevant for the port of this event. Therefore, for each port, a flag is hard coded in the related code hooks in order to indicate whether the port is concerned by at least one pattern. This simple mechanism avoids useless calls to the tracer driver.

5.4 Examples of pattern automata

Fig. 12 shows the internal representation of 5 patterns including the ones presented in Section 3.3. The 14 ports are represented on the left-hand side. The irrelevant ports are in italic. The relevant ports are linked to their corresponding patterns. Only two automata

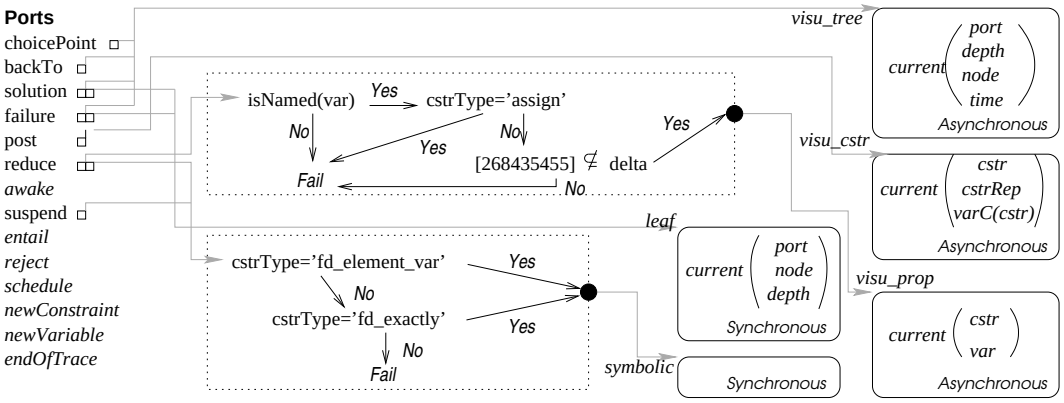


Figure 12: Internal representation of 5 simultaneous patterns

are necessary since three patterns check the port only. A set of actions is assigned to each automaton. This set is attached to "synchronous" or "asynchronous" and the label of the pattern.

5.5 Incremental Pattern Management

Since each active pattern is a specific automaton (or a list of specific automata when split), the `add` primitive has just to compile the new n patterns into m automata ($m \geq n$), linked them with their respective ports and store the labels with the lists of resulting automata. The `remove` primitive has just to delete the automata associated to the specified labels and to erase the dead links. After each operation, the port-filtering flags are updated so as to take into account the new state of the active patterns.

6 Prototype Implementation

In this Section, we briefly present the prototype implementation. In particular, in order for the overall architecture to be efficient, it is essential that the tracer is lazy. Trace information must not be computed if it is not explicitly required by a pattern. Indeed, an execution has many events and events potentially have many attributes. Most of them are not straightforwardly available, they have to be computed from the execution state or from the debugging data of the tracer. Systematically computing all the attributes at all the execution events would be terribly ineffective.

Fortunately, not all the attributes need to be computed at each event. According to the active patterns, only a subset of the attributes is needed: firstly, the attributes necessary to check the relevance of the current event with respect to the patterns, and secondly, the attributes requested by the patterns in case of matching. Therefore, the tracer must not

compute any trace attribute while none is needed. When a specific attribute is needed, it is computed and its value is stored until the end of the checking of the current event. If an attribute is used in several conditions, it is computed only once.

The tracer implemented in the current prototype, Codeine, strictly follows this guide line. Some core tracer mechanisms are needed to handle the debugging information (see [15] for more details). As shown in Sec. 7, the overhead induced by these mechanisms is marginal, even though constraint solvers do manipulate large and complex data.

Currently, Codeine is implemented in 6800 lines of C including comments. The tracer driver, including the communication mechanisms, is 1700 lines of C. The codeine tracer including the tracer driver is available under GNU Public Licence. The set of debugging and visualization tools called Pavot has been developed by Arnaud Guillaume and Ludovic Langevine. Both systems are available on line[2].

7 Experimental Results

This section assesses the performances of the tracer driver and its effects on the cost of the trace generation and communication. It shows several things. The overhead of the core tracer mechanisms is small, therefore the core tracer can be permanently activated. The tracer driver overhead is acceptable. There is no overhead for parallel search of patterns. The tracer driver overhead is predictable for given patterns. The tracer driver approach that we propose is more efficient than sending over a default trace, even to construct sophisticated graphical views. Answering queries is orders of magnitude more efficient than displaying traces. There is no need to a priori restrict the trace information. The performance of our tool is comparable to the state-of-the-practice while being more powerful and more generic.

7.1 Methodology of the Experiments

When tracing a program, some time is spent in the program execution (T_{prog}), some time is spent in the core mechanisms of the tracer (Δ_{tracer}), some time is spent in the tracer driver (Δ_{driver}), some time is spent generating the requested trace and sending it to the analysis process (Δ_{gcom}), some time is spent in the analyses (Δ_{ana}). Hence, if we call T the execution time of a traced and analysed program, we approximatively have:

$$T \simeq T_{prog} + \Delta_{tracer} + \Delta_{driver} + \Delta_{gcom} + \Delta_{ana}.$$

The mediator is a simple switch. The time taken by its execution is negligible compared to the time taken by the simplest analysis, namely the display of trace information. Trace analysis takes a time which vary considerably according to the nature of the analysis. The focus of this article is not to discuss which analyses can be achieved in reasonable time but to show that a flexible analysis environment can be offered at a low overhead. Therefore, in the following measurements $\Delta_{ana} = 0$.

[2] They can be retrieved at http://contraintes.inria.fr/~langevin/codeine/ and http://contraintes.inria.fr/~arnaud/pavot/.

Program	evts (10^6)	Trace Size (Gb)	T_{prog} (ms)	ε (ns)	$R_{tr.}$	Dev. for T_x in %
bridge	0.2	0.1	14	72	1.21	≤ 0.4
queens(256)	0.8	1.5	173	210	1.14	≤ 0.2
magic(100)	3.2	1.4	215	66	1.03	≤ 0.2
square(24)	4.2	20.8	372	88	1.05	≤ 0.6
golombF	15.5	3.4	7,201	464	1.01	≤ 0.4
golomb	38.4	7.9	1,721	45	1.00	≤ 0.5
golfer(5,4,4)	61.0	>30	3,255	53	1.05	≤ 0.7
propag	280.0	>30	3,813	14	1.28	≤ 1.0
queens(14)	394.5	>30	17,060	43	1.08	≤ 0.4

Table 1: Benchmark Programs and tracer overhead

7.2 Experimental setting

The experiments have been run on a PC, with a 2.4 GHz Pentium IV, 512 Kb of cache, 1 GB of RAM, running under the GNU/Linux 2.4.18 operating system. The last stable release (1.2.16) of GNU-Prolog has been used. The tracer is an instrumentation of the source code of this very same version and has been compiled in the same conditions by `gcc-2.95.4`. The execution times have been measured with the GNU-Prolog profiling facility whose accuracy is 1 ms. The measured executions consist of a batch of executions such that each measured time is at least 20 seconds. The measured time is the sum of *system* and *user* times. Each experimental time given below is the average time of a series of ten measurements. In each series, the maximal relative deviation was smaller than 1 %.

7.3 Benchmark programs

The 9 benchmark programs[3] are listed in Table 1, sorted by increasing number of trace events. Magic(100), square(4), golomb(8) and golfer(5,4,4) are part of CSPLib, a benchmark library for constraints by Gent and Walsh [9]. The golomb(8) program is executed with two strategies which exhibit very different response times. Those four programs have been chosen for their significant execution time and for the variety of constraints they involve. Four other programs have been added to cover more specific aspects of the solver mechanisms: Pascal Van Hentenryck's bridge problem (version of [6]); two instances of the n-queens problem; and "propag", the proof of infeasibility of $1 \leq x, y \leq 70000000 \wedge x < y \wedge y < x$.

The benchmark programs have executions large enough for the measurements to be meaningful. They range from 200,000 events to about 400 millions events. Furthermore, they represent a wide range of CLP(FD) programs.

The third column gives the size of the traces of the benchmarked programs for the default trace model. All executions but the smallest one exhibit more than a gigabyte, for executions sometimes less than a second. It is therefore not conceivable to systematically

[3]Their source code is available at http://contraintes.inria.fr/~langevin/codeine/benchmarks

generate such an amount of information. As a matter of fact measuring these size took us hours and, in the last three cases, exhausted our patience! Note that the size of the trace is not strictly proportional to the number of events because the attributes collected at each type of events are different. For example, for domain reductions, several attributes about variables, contraints and domains are collected while other types of events simply collect the name of the corresponding contraint.

The fourth column gives T_{prog}, the execution time in ms of the program simply run by GNU-Prolog. The fifth column shows the average time of execution per event $\varepsilon = \frac{T_{prog}}{\text{Nb. evt.}}$. It is between 14 ns and 464 ns per event. For most of the suite ε is around 50ns. The three remarkable exceptions are *propag* ($\varepsilon = 14$ ns), *queens(256)* ($\varepsilon = 210$ ns) and *golombF* ($\varepsilon = 464$ ns). The low ε is due to the efficiency of the propagation stage for the constraints involved in this computation. The large εs are due to a lower proportion of "fine-grained" events.

7.4 Tracer overhead

Table 1 also gives the results of the measurements of the overhead of the core tracer mechanisms. The measure of

$$T_{tracer} \simeq T_{prog} + \Delta_{core\_tracer}$$

is the execution time of the program run by the tracer without any pattern activated. The tracer maintains its own data for all events. However, no attribute is calculated and no trace is generated. The sixth column gives the ratio

$$R_{tr(acer)} = \frac{T_{tracer}}{T_{prog}}.$$

The seventh column gives the maximum deviation for T_{prog} and T_{tracer}.

Core tracer mechanisms can be permanently activated For all the measured executions R_{tracer} is less than 30% in the worst case, and less than 5% for five traced programs.

The results for R_{tracer} are very positive, they mean that the core mechanisms of the tracer can be systematically activated. Users will hardly notice the overhead. Therefore, while developping programs, users can directly work in "traced" mode, they do not need to switch from untraced to traced environments. This is a great confort. As soon as they will need to trace they can immediately get information.

7.5 Tracer driver overhead

The measure of

$$T_{driver} \simeq T_{prog} + \Delta_{core\_trace} + \Delta_{driver}$$

1a. *when port=post and isNamed(cname)*
do current(port,chrono,cident).
2a. *when port=reduce and*
(isNamed(vname) and isNamed(cname))
do current(port,chrono,cident).
3a. *when chrono=0 do current(chrono).*
4a. *when depth=50000 or (chrono>=1 and node=9999999)*
do current(chrono,depth).
5a: patterns 1a, 2a, 3a and 4a activated in parallel.

Figure 13: Patterns used to measure the tracer driver overhead

is the execution time of the program run by the tracer with the filtering procedure activated for generic patterns. Only the attributes necessary for the requested patterns are calculated at relevent events. In order for Δ_{gcom} to be zero, the patterns are designed such that no event matches them. One run is done per pattern. The patterns are listed in Figure 13. Pattern **1a** is checked on few events and on one costly attribute only. Pattern **2a** is checked on numerous events and on two costly attributes. Pattern **3a** is checked on all events and on one cheap attribute. Pattern **4a** is checked on all events and systematically on three attributes.

Tracer driver overhead is acceptable Figure 14 gives the results of the measurements of the overhead of the tracer driver for all the benchmark programs and for five patterns. The figure draws

$$R_{driver} = \frac{T_{driver}}{T_{prog}},$$

compared to the average time per event (ε) for the 5 patterns.

For all but one program, R_{driver} is negligible for the very simple patterns and less than 3.5 for pattern **5a** which is the combination of 4 patterns. For programs with a large ε, even searching for pattern **5a** is negligible. In the worst case, an overhead of 8 is still acceptable.

No overhead for parallel search of patterns When n patterns are checked in parallel they already save
$(n-1)T_{tracer}$ compared to the search in sequence which requires to executes n times the program instead of one time. Figure 14 further shows that

$$\Delta^{1a}_{driver} + \Delta^{2a}_{driver} + \Delta^{3a}_{driver} + \Delta^{4a}_{driver} > \Delta^{(1|2|3|4)a}_{driver}.$$

As a matter of fact, the curve $\Sigma R = R_1 + R_2 + R_3 + R_4 - 3$, that adds the overheads of the four separated patterns, is above the curve of R^{5a}_{driver}. This means that not only is there no overhead in the filtering mechanism induced by the parallel search, but there is even a minor gain.

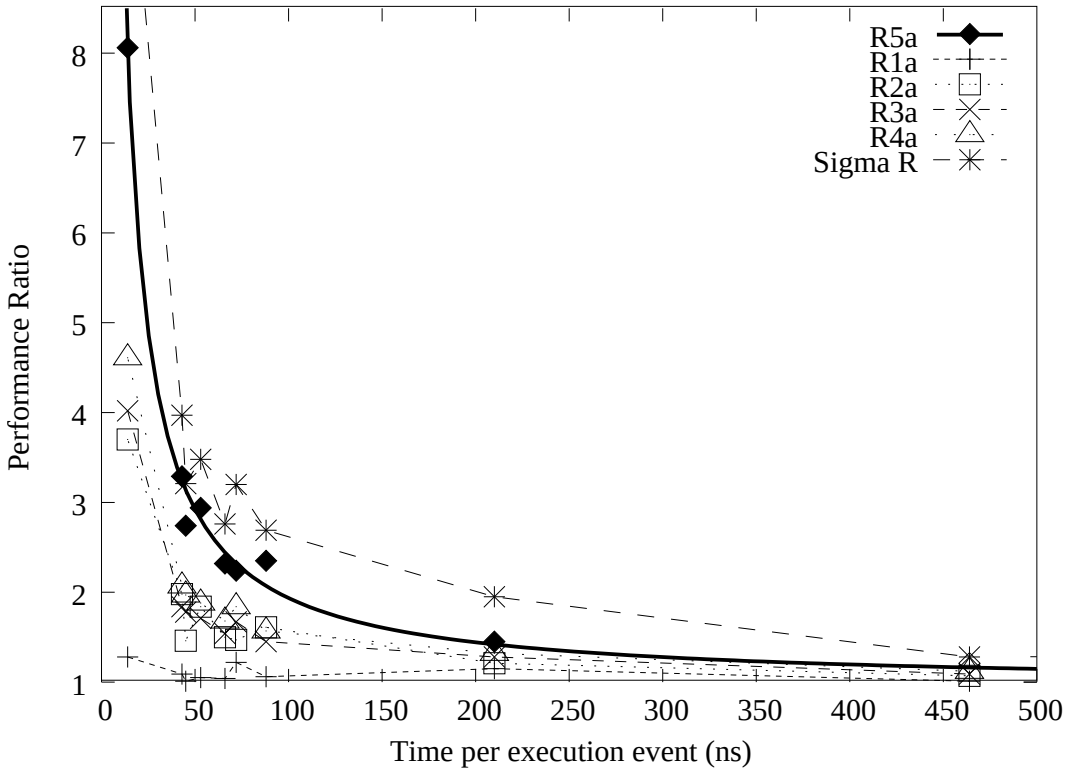


Figure 14: Cost of the tracer driver mechanisms compared to ε

Tracer driver overhead is predictable The measured points of Figure 14 can be interpolated with curves of the form

$$R_{driver} = a + b/\varepsilon.$$

Figure 15 recalls the curve for pattern **5a** and gives the number of events. It shows that there is no correlation between the size of the trace and the tracer driver overhead.

Those results mean that the tracer and tracer driver overheads per event can be approximated to constants depending on the patterns and *independant of the traced program.* Indeed, let us assume that that $\Delta_{core\_trace} = N\delta_{core\_trace}$ and $\Delta_{driver} = N\delta_{driver}$ where N is the number of events of an execution, δ_{core_trace} and δ_{driver} are the average time per event taken respectively by the core tracer mechanism and the tracer driver, for all the programs. We have also already assumed that

$$T_{driver} \simeq T_{prog} + \Delta_{core\_trace} + \Delta_{driver},$$

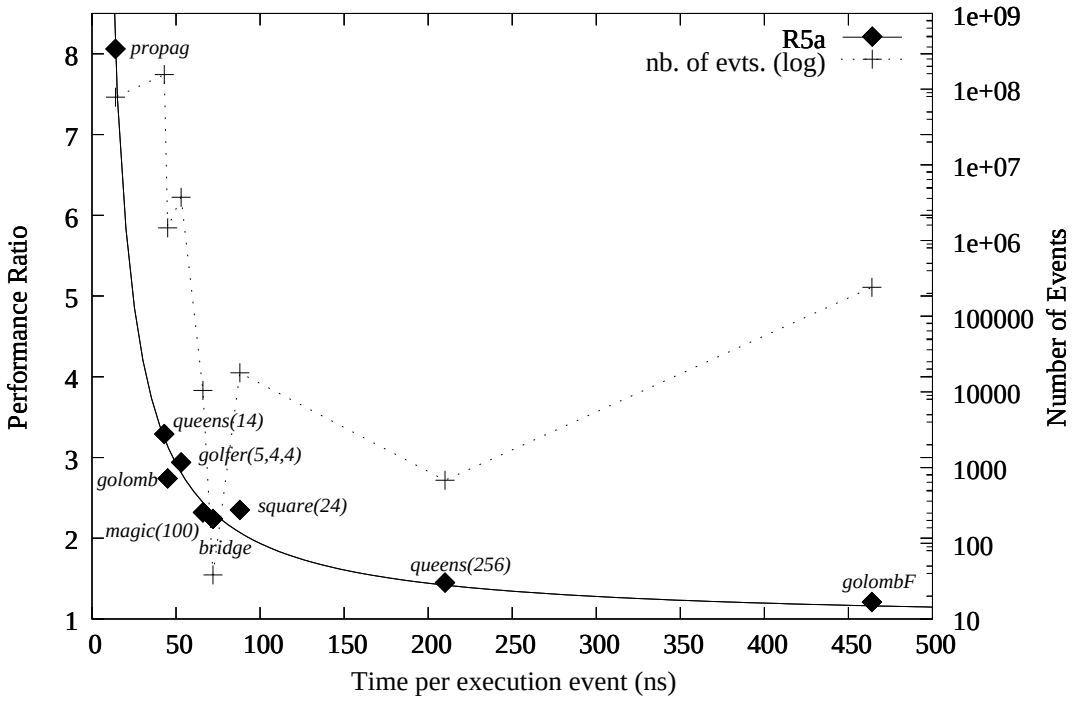


Figure 15: Cost of the tracer driver mechanisms compared to the number of events

and we have

$$R_{driver} = \frac{T_{driver}}{T_{prog}}, \quad \text{and} \quad T_{prog} = N\varepsilon$$

therefore

$$R_{driver} \simeq \frac{T_{prog} + \Delta_{core\_trace} + \Delta_{driver}}{T_{prog}}$$

$$R_{driver} \simeq 1 + \frac{N(\delta_{core\_trace} + \delta_{driver})}{N\varepsilon}$$

$$R_{driver} \simeq 1 + \frac{\delta_{core\_trace} + \delta_{driver}}{\varepsilon}$$

The measured R_{driver} for pattern **5a** is

$$R_{driver} = 0.95 + \frac{98.58}{\varepsilon}.$$

1b. *cstr: when port=post do current(chrono,cident,cinternal).*
tree: when port in [failure,backTo, choicePoint,solution] do current(chrono,node,port).
2b. *newvar: when port=newVariable do current(chrono, vident, vname).*
dom: when port in [choicePoint,backTo,solution] do current(chrono,node,port,named_vars,full_dom).
3b. *propag1: when port=reduce do current(chrono).*
4b. *propag2: when port=awake do current(chrono).*

Figure 16: Event patterns used to assess the trace generation and the communication overhead

For pattern **5a**, the average time per event taken by the core tracer mechanism and the tracer driver ($\delta_{core\_trace} + \delta_{driver}$) can therefore be approximated to $98.58ns$.

The R_{driver} overhead could thus be made predictable. For a given program, it is easy to automatically measure ε, the average time of execution per event. For a library of patterns $\delta_{core\_trace} + \delta_{driver}$ can be computed for each pattern. We have shown above that the overhead of the search in parallel of different patterns can be over approximated by the sum of all the overheads. Our environment could therefore provide estimation mechanisms. When ε would be too small compared to $\delta_{core\_trace} + \delta_{driver}$ the user would be warned that the overhead may become large.

7.6 Communication overhead

The measure of

$$T_{gcom} \simeq T_{core\_tracer} + \Delta_{driver} + \Delta_{gcom}$$

is the execution time of the program run by the tracer. A new set of patterns are used so that some events match the patterns, the requested attributes of the matched events are generated and sent to a degenerated version of the mediator: a C-program that simply reads the trace data on its standard input. Due to lack of space we only show the result of program golomb(8) which has a median number of events and has a median ε.

The patterns are listed in Figure 16. Pattern **1b**, composed of two basic patterns, allows a "bare" search tree to be constructed, as shown by most debugging tools. Pattern **2b** (two basic patterns) allows the display of 3D views of variable updates as shown in Figure 8. Pattern **3b** and pattern **4b** provide two different execution details to decorate search trees. Depending on the tool settings, three different visual clues can be displayed. One is shown in Figure 8.

Table 2 gives the results for the above patterns and some of their combinations. All combinations correspond to existing tools. For example, combining **1b** with **3b** or/and **4b** allows a Christmas tree as shown in Figure 8 to be constructed with two different parameterizations. The 2$^{\text{nd}}$ column gives the number of events which match the pattern. The 3$^{\text{rd}}$ column gives the size of the resulting XML trace as it is sent to the tool. The 4$^{\text{th}}$ col-

Program: golomb(8) $\varepsilon = 45$ns $T_{prog=1.73s}$					
Patterns	Traced evts (10^6)	Trace size (M bytes)	Elapsed time (s)	$R_{dr.}$	R_{gcom}
1b	0.36	21	4.50	1.03	2.6
2b	0.13	111	16.17	1.02	9.35
3b	5.04	141	33.57	1.14	19.40
4b	14.58	394	89.40	1.32	51.68
(1|2)b	0.36	124	17.47	1.04	10.09
(1|3)b	5.40	162	36.08	1.15	20.85
(1|4)b	14.94	415	92.71	1.33	53.59
(1|3|4)b	19.97	556	122.72	1.44	70.93
(1|2|3|4)b	19.97	660	136.80	1.44	79.07
def. trace	38.36	7,910	393.08	1.96	227.21

Table 2: Cost of the trace generation and communication

umn gives the elapsed time[4]. The 5$^{\text{th}}$ column gives the ratio R_{driver}, recomputed for each pattern. The 6$^{\text{th}}$ column gives the ratio $R_{gcom} = \frac{T_{gcom}}{T_{prog}}$.

Filtered trace is more efficient and more accurate than default trace The last line gives results for the *default* trace. The *default* trace contains twice as many events as the trace generated by pattern **(1|2|3|4)b**, but it contains more attributes than requested by the pattern; As a result, its size is ten times larger because and its R_{gcom} overhead is three times larger. At the same time, it does not contain all the attributes. In that particular case, some relevant attributes are missing in the default trace while there are present in the trace generated by pattern **(1|2|3|4)b**. The attributs not sent by the default trace can be reconstructed by the analysis module, but this requires further computation and memory resources.

As a consequence, the tracer driver approach that we propose is more efficient than sending over a default trace, even to construct sophisticated graphical views. The accuracy and the lower volume of the trace ease its post-processing by the debugging tools.

Answering queries is more efficient than displaying traces R_{gcom} is always much larger that R_{driver}, from 2.6 to 79.07 in our exemple. Therefore, queries using patterns that drastically filter the trace have significantly better response time than queries that first display the trace before analysing it.

When debugging, programmers often know what they want to check. In that case they are able to specify queries that demand a simple answer. In such a case our approach is significantly better than sending trace information to an analyzer.

[4]Here system and user time are not sufficient because two processes are at stake. T_{prog} has been re-measured in the same conditions.

No need to a priori restrict the trace information Many tracers restrict a priori the trace information in order to reduce the volume of trace sent to an analyzer. This restrict the possibilities of the possible dynamic analysis without preventing the big size and time overhead as shown above with the default trace which does not contain important information while being huge.

With our approach, trace information which is not requested does not cost much, therefore our trace model can afford to be much very rich. This enlarges the possibility of adding new dynamic anayses.

Performance are comparable to the state-of-the-practice R_{gcom} varies from 2.6 to 79.07. To give a comparison the Mercury tracer of Somogyi and Henderson [18] is regularly used by Mercury developers. For executions of size equivalent to those of our measurements, the Mercury tracer overhead has been measured from 2 to 15, with an average of 7 [12]. Hence the ratios for patterns **1b**, **2b** and **1|2b** are quite similar to the state-of-the-practice debuggers. The other patterns show an overhead that can discourage interactive usage. However, these patterns are more thought of for monitoring than debugging when the interaction does not have to be done in real time. Note, furthermore, that for the measured programs, the absolute response time is still on the range of two minutes for the worst case. When debugging, this is still acceptable.

Our approach allows therefore to have the tracer present but idle by default. When a problem is encountered, simple queries can be set to localize roughly the source of the problem. Then, more costly patterns can be activated on smaller parts of the program. This is pretty much like what experienced programmers do. The difference with our approach is that they do not have to change tools, neither to reset the parameterizations of the debugger.

8 Related Work

Kraut [4] implements a finite state machine to find sequences of execution events that satisfy some patterns, called *path rules*. Several patterns are allowed and they can be enabled or disabled during the execution, using a labeling policy. Specified actions are triggered when a rule is satisfied but they are limited to some debugger primitives, such as a message display or a counter increasing. The main interest of this tool is to abstract the trace and to allow the easy development of monitors. The trace analysis is necessarily synchronous and does not benefit from the power of a complete programming language.

Reiss and Renieris [17] have an approach similar to ours. They also structure their dynamic analyses into three different modules: 1) extraction of trace, 2) compaction and filtering and 3) visualization. They provide a number of interesting compaction functions which should be integrated in a further version of our system. They, however, first dump the whole trace information in files before any filtering is processed. With our tracer driver filtering is done on the fly, and section 7 has shown that this is much more efficient than first storing in files.

Coca [8] and Opium [8] provide a trace query mechanism, respectively for C and Prolog. This mechanism is synchronous and does not allow concurrent analysis. It can be easily emulated with our tracer driver and an analyzer mediator written in Prolog.

Hy$^+$ [5] writes the trace into a real relational database to query it with SQL. This is even slower than writing the trace simply into a file. However, when on the fly performance is not an issue, for example for post mortem analysis, this is a very powerful and elegant solution.

Dalek [16] is a powerful extension of *gdb*. It allows user to associate sequences of execution events to specific synchronous handlers written in a dedicated imperative language. This language includes primitives to retrieve additional trace-data and to synchronize the execution. The management of handlers is not incremental. A key feature of Dalek, especially useful in an imperative language, is the explicit "queue of events" that stores the achieved execution events. The user can explicitly remove events from this queue and add higher-level events. This approach requires an expensive storage of a part of the trace but enables both monitoring, debugging and profiling of programs.

In EBBA [3], expected program behaviors are modeled as relationships between execution events. Those models are then compared to the actual behavior during execution. EBBA tries to recognize relevant sequences of events and to check some constraints about such sequences. A kind of automaton is built to find instantiation of the models. The events are first generated by the tracer before being filtered according to the automata. Our approach allows to filter the execution event directly inside the tracer, this is more efficient. Nevertheless, EBBA recognizes sequences of events whereas we filter one event at time. Our approach could be used upstream of the sequence recognition. The incrementality of the event patterns could be used to adapt the relevant events to the states of the automata.

UFO [1] offers a more powerful language to specify patterns and monitors than EBBA. The patterns can involve several events, not necessarily consecutive. In our framework, the monitors have to be implemented in the analyzer with a general programming language. A further extension should allow at least to implement monitors in the trace driver to improve efficiency. UFO, however, does not allow the same flexibility as our tracer driver, and is heavier to use for interactive debugging.

So far, our framework applies only to a single execution and does not easily scale to compare numerous executions as is done in "batch mode" by Harrolds et al. [14]. It seems, however, possible to extend our framework so that two executions can be run in parallel with two tracer drivers. This would allow to implement the debugging analyses of Zeller et al [21] and Abramson et al. [19] which compare two executions at a given moment.

9 Conclusion

In this paper we have presented a tracer driver which allows both synchronous and asynchronous trace analysis in the same execution, fitting all the needs of the classical usages of a tracer into a single tool. We have defined an expressive language of event patterns where relevant events are described by first order formulæ involving most of the data the tracer can

access. Specific primitives enable the retrieval of large pieces of data "on demand" and the adaptation of the event patterns to the evolving needs of the trace analyzer. Therefore, the produced trace is accurate: trace generation, trace communication and trace post-processing are speeded up. The tracer driver provides a powerful front-end for complex debugging tools based on trace data.

Acknowledgment The authors thank Pierre Deransart and their OADymPPaC partners for fruitful discussions, as well as Guillaume Arnaud for his careful beta-testing of Codeine.

References

[1] M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. In W. Emmerich and D. Wile, editors, *Proceedings fo the 17th International Conference on Automated Software Engineering (ASE'02)*, pages 217–222. IEEE Press, 2002.

[2] T. Ball. The concept of dynamic analysis. In O. Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 1999.

[3] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, Feb. 1995.

[4] B. Bruegge and P. Hibbard. Generalized path expressions: A high-level debugging mechanism. *The Journal of Systems and Software*, 3:265–276, 1983. Elsevier.

[5] M. Consens, M. Hasan, and A. Mendelzon. Visualizing and querying distributed event traces with Hy+. In W. Litwin and T. Risch, editors, *Applications of Databases, First International Conference*, pages 123–141. Springer, Lecture Notes in Computer Science, Vol. 819, 1994.

[6] D. Diaz. GNU prolog, a free prolog compiler with constraint solving over finite domains, 2003. http://gprolog.sourceforge.net/ Distributed under the GNU license.

[7] M. Ducassé. Coca: An automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–513. ACM Press, May 1999.

[8] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 39:177–223, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

[9] I. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of CP-99.

[10] S. Hackstadt and A. Malony. DAQV: Distributed Array Query and Visualization Framework. *Theoretical Computer Science*, 196(1–2):289–317, Apr. 1998.

[11] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90. ACM Press, 1998.

[12] E. Jahier and M. Ducassé. Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming*, 2(4-5):611–643, July-September 2002.

[13] C. Jeffery and R. Griswold. A framework for execution monitoring in icon. *Software-Practice and Experience*, 24(11):1025–1049, November 1994.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press, 2002.

[15] L. Langevine, M. Ducassé, and P. Deransart. A propagation tracer for Gnu-Prolog: from formal definition to efficient implementation. In C. Palamidessi, editor, *Proc. of the 19th Int. Conf. in Logic Programming*. Springer-Verlag, Lecture Notes in Computer Science 2916, December 2003.

[16] R. Olsson, R. Crawford, and W. Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the Summer 1990 USENIX Conference: June 11–15, 1990*, pages 221–232, 1990.

[17] S. Reiss and M. Renieris. Encoding program executions. In M.-J. Harrold and W. Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Press, 2001.

[18] Z. Somogyi and F. Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, volume 30(4). Elevier, Electronic Notes in Theoretical Computer Science, 1999. http://www.elsevier.nl/cas/tree/store/tcs/free/entcs/store/tcs30/cover.sub.sht.

[19] R. Sosic and D. Abramson. Guard: A relative debugger. *Software - Practice and Experience*, 27(2):185–206, 1997.

[20] R. Wilhelm and D. Maurer. *Compiler design*. Addison-Wesley, 1995. ISBN: 0-201-42290-5.

[21] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Contents

1 Introduction **3**
1.1 A Tracer Driver to Efficicently Share Instrumentations 3
1.2 Tracer and Analyzers Interactions 4
1.3 Contributions 5

2 Overview of the tracer driver **6**

3 Event patterns **8**
3.1 Trace events 8
3.2 Patterns 9
3.3 Examples of patterns 10

4 Analyzer mediator **11**

5 Filtering mechanism **12**
5.1 Tracer driver algorithm 13
5.2 Pattern automata 13
5.3 Specialization thanks to the port 14
5.4 Examples of pattern automata 14
5.5 Incremental Pattern Management 15

6 Prototype Implementation **15**

7 Experimental Results **16**
7.1 Methodology of the Experiments 16
7.2 Experimental setting 17
7.3 Benchmark programs 17
7.4 Tracer overhead 18
7.5 Tracer driver overhead 18
7.6 Communication overhead 22

8 Related Work **24**

9 Conclusion **25**

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399